ARL-TR-7474 • Sep 2015

US Army Research Laboratory

Using the Gilbert-Johnson-Keerthi Algorithm for Collision Detection in System Effectiveness Modeling

by Benjamin A Breech

NOTICES

Disclaimers

The findings in this report are not to be construed as an official Department of the Army position unless so designated by other authorized documents.

Citation of manufacturer's or trade names does not constitute an official endorsement or approval of the use thereof.

Destroy this report when it is no longer needed. Do not return it to the originator.

ARL-TR-7474 • SEP 2015

US Army Research Laboratory

Using the Gilbert-Johnson-Keerthi Algorithm for Collision Detection in System Effectiveness Modeling

by Benjamin A Breech
Weapons and Materials Research Directorate, ARL

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED (From - To)
September 2015	Final	May 2013 - August 2014

4. TITLE AND SUBTITLE
Using the Gilbert-Johnson-Keerthi Algorithm for Collision Detection in System Effectiveness Modeling

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)
Benjamin A Breech

5d. PROJECT NUMBER
AH80

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
U.S. Army Research Laboratory
ATTN: RDRL-WML-A
Aberdeen Proving Ground, MD 21005-5066

8. PERFORMING ORGANIZATION REPORT NUMBER
ARL-TR-7474

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSOR/MONITOR'S ACRONYM(S)

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

13. SUPPLEMENTARY NOTES
Author email: <benjamin.a.breech.civ@mail.mil>

14. ABSTRACT
I present an overview of the Gilbert-Johnson-Keerthi (GJK) algorithm for collision detection using a geometrical approach that relies on using vector cross and dot products to determine if a collision has occurred. While this geometrical approach may be more intuitive and easier to understand than the original algebraic approach, it also requires careful implementation to avoid easy-to-make mistakes. Finally, The results of the GJK algorithm are compared to a different algorithm for determining collisions among triangles. Overall, GJK is well suited for use in system effectiveness modeling in order to determine when 2 objects in system simulation collide.

15. SUBJECT TERMS
threat armor interaction, modular modeling

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			Benjamin A Breech
Unclassified	Unclassified	Unclassified	UU	40	19b. TELEPHONE NUMBER (Include area code) 410-278-3976

Standard Form 298 (Rev. 8/98)
Prescribed by ANSI Std. Z39.18

Contents

List of Figures **iv**

1 Introduction **1**

2 Mathematical Preliminaries **2**

2.1 Dot and Cross Products 2

2.2 Convex Sets and Convex Hulls 3

2.3 Minkowski Sum (and Difference) 6

2.4 Support Functions 7

2.5 Simplices 8

3 GJK Algorithm **9**

3.1 Basic Operation of GJK 9

3.1.1 Overview of GJK 9

3.1.2 Two Examples of GJK Operating 9

3.1.3 Termination Conditions 11

3.1.4 GJK Algorithm 13

3.2 Simplex Processing 14

3.2.1 Processing a 0-simplex 16

3.2.2 Processing a 1-simplex 16

3.2.3 Processing a 2-simplex 19

3.2.4 Processing a 3-simplex 22

3.3 GJK Efficiency: Qualitative 28

3.4 GJK Summary 29

4 Verification of Implementation **30**

5 Conclusion **31**

6 References and Notes **32**

Distribution List **33**

List of Figures

Fig. 1 Examples of convex and non-convex objects. A circle is not convex because any chord is not in the set. Similarly, a solid crescent is not convex because a line segment for points in the wedge area are not included. A solid disk is convex. ……4

Fig. 2 Sketch of the rubber band analogy for computing convex hulls. We stretch a rubber band around a set of points and let the rubber band go. The band will snap around the outermost points, forming the convex hull. ……5

Fig. 3 Two example triangles and their Minkowski Difference ……6

Fig. 4 Two example Mdiff sets demonstrating how GJK operates. The set on the left includes the origin while the set on the right does not (the origins are indicated by the small plus signs). The dashed lines indicate the segments added to the simplex by GJK. …… 10

Fig. 5 Closeup view of moving from point A to point E for the examples from Fig. 4. For both cases, GJK computed the direction $\mathbf{d}$ to search for a new point, and, through the support function, found point E. On the left, the origin was crossed while moving from A to E, while on the right the origin was not crossed. …… 12

Fig. 6 Sketch of a 1-simplex, which is a line segment. The origin could be in any of the 4 indicated regions. …… 17

Fig. 7 Sketch of a 2-simplex and the regions created around it. …… 19

Fig. 8 Sketch showing a 3-simplex (a tetrahedron). The point A lies above the plane determined by the triangle $\triangle BCD$. The point B lies behind the $\triangle ACD$ triangle face. …… 24

1. Introduction

The Gilbert-Johnson-Keerthi[1] (GJK) algorithm is a very efficient algorithm for determining the minimum distance between 2 convex objects. GJK also doubles as a very fast collision detection, which has proven extremely useful in modeling work.[2] In particular, while modeling the system effectiveness of different threats, sensors, and armor packages, we required a way of very quickly determining if a threat and counter-munition had collided, if a threat had impacted a vehicle, or if multiple counter-munitions had collided, and so on. All of these problems required a very fast algorithm to determine if 2 objects intersected where the objects may not be simple lines, cubes, cylinders, or other polygons.

While GJK was useful in our work, implementing GJK proved to be extremely difficult. The original GJK paper[1] takes an algebraically intensive approach to compute results, which becomes difficult to implement. Alternatively, GJK can be implemented using more geometrical concepts, which creates a more intuitive and understandable algorithm. Numerous tutorials have been posted[3–9] adopting the geometrical approach. However, we encountered the following difficulties:

- The tutorials do not agree with each other. The basic concepts are the same across various tutorials, but the implementation details conflict with each other. Certain cases may be ignored or handled inconsistently.
- Most tutorials only focus on 2 dimensions. The 3-dimensional (3-D) case, however, is not a straightforward generalization of the 2-dimensional (2-D) case.
- Very little is written about the assumptions involved in the geometrical case. This leads to difficulties since certain portions of the algorithm require very specific orderings.

These problems may not be recognized by the tutorial authors themselves since GJK is an iterative algorithm that improves results from one iteration to the next until an answer is found. The iterative nature of GJK can mask many bugs in an implementation because the implementation can self-correct in later iterations. That is, if a bug exists in a particular case, the implementation may cause GJK to search for results in a wrong direction. However, later iterations could eventually bring the implementation back to the correct direction. Thus, an implementation can have a

bug, but still pass many test cases, resulting in the bug staying in the code for a long time.

These problems have led us to record the details of our implementation of the GJK algorithm. We discuss the assumptions going into each portion of the implementation, as well as why certain orderings are required. Given the difficulties above, we cannot state that our implementation is bug free. However, we do discuss our efforts at verifying our implementation against an outside source.

2. Mathematical Preliminaries

We first discuss a few mathematical concepts that are needed later. Readers familiar with this material can simply pass to the next section.

We use the notation that A refers to a point in space, while $\mathbf{A}$ refers to the vector from the origin to the point A. $\mathbf{AB} = \mathbf{B} - \mathbf{A} =$ vector from point A to point B. We use A_i to refer to the i-th component of vector A. We say that $\mathbf{A}$ is a zero vector if all $A_i = 0$.

Points and vectors may be gathered into sets, which we indicate with script letters, e.g., $\mathscr{A}$, $\mathscr{B}$. Two points can also form a line segment, which we indicate as $\overline{AB}$. Three (non-colinear) points form a triangle, which we indicate as $\triangle ABC$.

2.1 Dot and Cross Products

The (scalar) dot and (vector) cross product are the 2 most basic product operations involving vectors. The dot product in Cartesian coordinates is

$$\mathbf{A} \cdot \mathbf{B} = \sum_i A_i B_i, \tag{1}$$

Geometrically, $\mathbf{A} \cdot \mathbf{B} > 0$ means that $\mathbf{A}$ has a component in the same direction as $\mathbf{B}$ (and vice-versa). Informally, we can say that the 2 vector point roughly in the same direction if the dot product is positive. $\mathbf{A} \cdot \mathbf{B} < 0$ means the vectors point in opposite directions. $\mathbf{A} \cdot \mathbf{B} = 0$ means the 2 vectors are perpendicular provided that $\mathbf{A}$ and $\mathbf{B}$ are not zero vectors.

The cross product is vector whose i-th component is defined as

$$(\mathbf{A} \times \mathbf{B})_i = \varepsilon_{ijk} A_j B_k, \tag{2}$$

where ε_{ijk} is the Levi-Civita symbol whose value is 1 if ijk is an even permutation (i.e., 123, 231, or 312), -1 if ijk is an odd permutation (i.e., 321, 213, or 132), and 0 otherwise. The x component of the cross product is thus $A_yB_z - A_zB_y$. Geometrically, $\mathbf{A} \times \mathbf{B}$ is a vector that is perpendicular to both $\mathbf{A}$ and $\mathbf{B}$. This interpretation is very important in Cartesian coordinates where 2 linearly independent vectors define a plane. In this case, the cross product of the vectors would define a vector normal to the plane.

For the dot product, the order of the vectors is unimportant since $\mathbf{A} \cdot \mathbf{B} = \mathbf{B} \cdot \mathbf{A}$. This is not true for the cross product where changing the order inverts the sign, i.e., $\mathbf{A} \times \mathbf{B} = -\mathbf{B} \times \mathbf{A}$. Geometrically, this implies that a plane has 2 unit normals that point in opposite directions. This is very important for the GJK algorithm where we take the cross products in very particular orders so that the normals point in the direction we require.

In some instances, we require the vector cross product of 3 vectors, $\mathbf{A}, \mathbf{B}$, and $\mathbf{C}$. Parentheses may be used to specify which cross product should be performed first. If no parentheses are given, then the cross products proceed from left to right. That is, $\mathbf{A} \times \mathbf{B} \times \mathbf{C}$ is evaluated as $(\mathbf{A} \times \mathbf{B}) \times \mathbf{C}$.

2.2 Convex Sets and Convex Hulls

GJK works on convex objects (and sets). Let $\mathscr{A}$ be a set and let P_1 and P_2 be any 2 points in $\mathscr{A}$. If all the points on the shortest path (a line in Euclidean geometry) between P_1 and P_2 are also in $\mathscr{A}$, then $\mathscr{A}$ is a convex set.

Figure 1 shows some examples of convex sets. A circle is not convex because it does not include any interior points. Thus, any chord of the circle is not included in the set, which means the set cannot be convex. By contrast, a solid disk is convex because the disk includes the interior points. In general, any hollow object (circle, empty cube, etc.) is not convex because the interior points are not included. Only solid, "filled in," objects can be convex. Note, however, that not all solid objects are convex as the crescent in Fig. 1 demonstrates.

GJK only works with convex sets and objects. This restriction means that GJK may not be able to determine if 2 objects with holes in them intersect. In practical terms, this restriction is rarely an inconvenience as most objects can be regarded as convex for the purposes of determining an intersection. This is true for our modeling

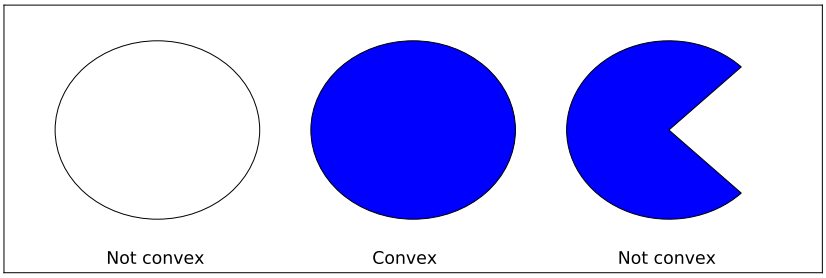

Fig. 1 Examples of convex and non-convex objects. A circle is not convex because any chord is not in the set. Similarly, a solid crescent is not convex because a line segment for points in the wedge area are not included. A solid disk is convex.

work, where we choose cylinders and other convex objects to represent threats and vehicles. This is also true for objects that do have holes in them provided those objects intersect with other objects larger than the holes. For example, consider a net thrown over a person. Strictly speaking, neither the net nor the person can be regarded as convex. The net has a lot of empty spaces between threads while the person could raise her limbs out from the body, creating empty spaces. However, for the purposes of throwing a net over the person, we can regard both as being convex provided the spaces in the net are too small to pass over the person. In such a scenario, the net will still drape over the person as if both were convex.

We do not necessarily need to keep track of all the points in the convex set $\mathscr{A}$. Since any line segment joining 2 points in $\mathscr{A}$ must also be in $\mathscr{A}$, there must be some minimal number of points on the outer edge of $\mathscr{A}$ such that we can draw line segments between those points and still recover all of $\mathscr{A}$. That is, there is some set $\mathscr{A}' \subseteq \mathscr{A}$ such that $\mathscr{A}'$ is convex and contains the region bounded by $\mathscr{A}$. The points in $\mathscr{A}'$ form the outer envelope (or hull) of $\mathscr{A}$. Since $\mathscr{A}'$ is, by construction, convex, we term this set the convex hull of $\mathscr{A}$, which we write as $\mathscr{A}' = \text{Conv}(\mathscr{A})$.

Figure 2 shows a useful visualization of a convex hull based on the rubber band analogy. Suppose we have a finite set of 2-D points. We first place a pin at each point and stretch a giant rubber band around all the pins. If we let go of the rubber band, it will collapse around the pins on the exterior of the set. The rubber band would then form the convex hull. This analogy also demonstrates that for any *finite* set of points, we can always determine a convex hull covering all the points.

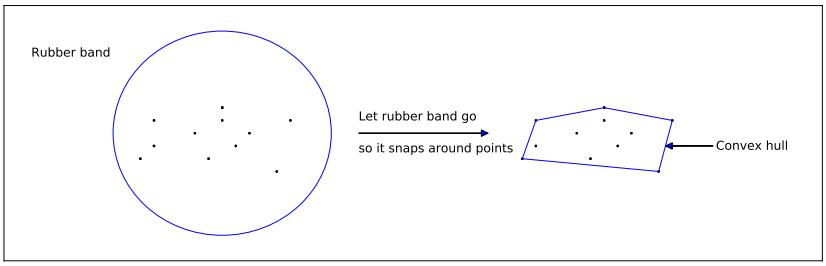

Fig. 2 Sketch of the rubber band analogy for computing convex hulls. We stretch a rubber band around a set of points and let the rubber band go. The band will snap around the outermost points, forming the convex hull.

For a solid disk, the convex hull contains all the points on the circumference of the disk, i.e., the bounding circle. This may be confusing at first since a circle, by itself, is not convex. However, the circle is the convex hull of a disk, which is convex. The crucial insight is to realize that the convex hull of a set bounds the original set. If a circle is the convex hull, then the original set included all the points on the circumference of the circle as well as the interior points, i.e., the solid disk.

A convex hull generally simplifies algorithms since we can analyze a smaller subset. A triangle, for example, is a convex object containing all the interior points and the perimeter connecting the vertices. Any analysis we perform would need to account for all those points. However, we can specify the convex hull of the triangle using only the 3 vertices of the triangle since those 3 points bound the triangle region. By using the convex hull, we only need to analyze 3 points. Algorithms, such as the Graham scan, exist to efficiently compute the convex hull of any finite set.

For a finite set of N points, the convex hull may involve up to N points. If N is very large, keeping all N points may be impractical. Depending on the application, we may not need to keep all the points. The theorem of Carathéodory states that given a set $\mathscr{A}$ consisting of points in d dimensions and a point $P \in \text{Conv}(\mathscr{A})$, there exists a subset $\mathscr{A}' \subseteq \mathscr{A}$ such that $\mathscr{A}'$ has, at most, $d+1$ points and $P \in \text{Conv}(\mathscr{A}')$. For 3 dimensions, this means we need, at most, 4 points from $\mathscr{A}$ to construct a convex hull containing a given P.

The algebraic approach to GJK involves computing convex hulls of small sets. The geometric approach to GJK does not directly compute the convex hulls. Instead, the

geometric approach uses convex hulls and the theorem of Carathéodory indirectly, as is seen in the next section.

2.3 Minkowski Sum (and Difference)

Suppose we have 2 sets, $\mathscr{A}$ and $\mathscr{B}$, consisting of vectors. We can form a new set by adding each vector in $\mathscr{A}$ to each vector in $\mathscr{B}$, and another new set by taking their difference. This is the definition of the Minkowski Sum and Difference sets. Formally,

$$\text{Msum}(\mathscr{A},\mathscr{B}) = \{\mathbf{X}+\mathbf{Y} \mid \mathbf{X} \in \mathscr{A} \wedge \mathbf{Y} \in \mathscr{B}\} \tag{3}$$

$$\text{Mdiff}(\mathscr{A},\mathscr{B}) = \{\mathbf{X}-\mathbf{Y} \mid \mathbf{X} \in \mathscr{A} \wedge \mathbf{Y} \in \mathscr{B}\} \tag{4}$$

Formally speaking, Msum and Mdiff only apply to sets of vectors. However, in Cartesian geometry, the concepts can be applied just as easily to points in space. Thus, we often use Msum and Mdiff with respect to points. Figure 3 shows an example of the Minkowski Difference for 2 triangles.

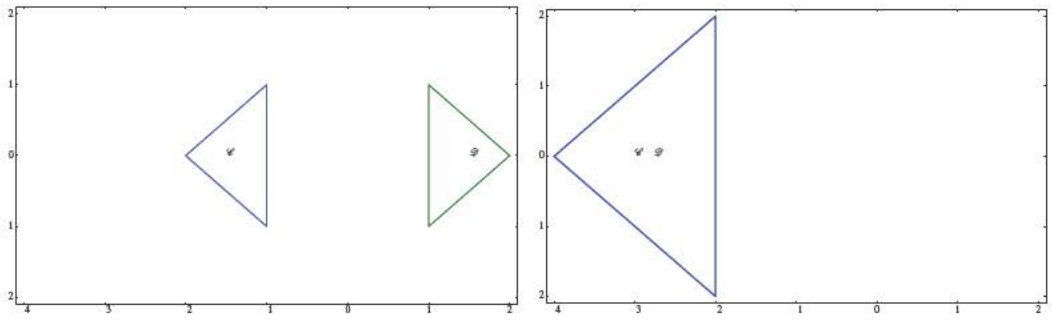

Fig. 3 Two example triangles and their Minkowski Difference

The Minkowski Difference has the property that if 2 sets intersect, then Mdiff will contain the origin. This is trivial to see since 2 sets intersecting means they must have at least 1 point in common, so Mdiff will include the origin. This property is important for GJK's operation.

Rather than simple sets, we are concerned with objects, such as a triangle, a sphere, or any convex polygon. The Minkowski Sum and Difference are still defined and are convex, provided the objects are convex. If the 2 objects intersect, the origin is contained within the Minkowski Difference. More specifically, the origin lies somewhere within the volume bounded by the convex hull of the Minkowski Difference.

GJK decides if 2 objects, $\mathscr{A}$ and $\mathscr{B}$, intersect by determining if Mdiff$(\mathscr{A},\mathscr{B})$ contains the origin or not. GJK does not need to compute the entire Minkowski Difference, which would be impractical for large objects or objects like spheres. Instead, GJK relies upon support functions to determine points on the convex hull of the Minkowski Difference.

2.4 Support Functions

The support function for a convex object takes a direction and returns the vector to a point in the object that is farthest in the given direction. Support functions do not have to return a unique point. Consider triangle $\mathscr{C}$ shown in Fig. 3. There is no vertex point exactly in the x direction, which means the support function could either interpolate between the points $(0,-1)$ and $(0,1)$, or simply return either of those vertices as both are equally far in the x direction.

If the object consists of a small number of points on the convex hull, such as the vertices of a triangle, then the support function simply returns a vector to the point whose dot product with the direction is maximal. Mathematically, for object $\mathscr{O}$ and direction $\mathbf{d}$,

$$\text{support}(\mathscr{O},\mathbf{d}) = \mathbf{P} \mid P \in \mathscr{O} \wedge \mathbf{P}\cdot\mathbf{d} \geq \mathbf{Q}\cdot\mathbf{d}\ \forall Q \in \mathscr{O} \tag{5}$$

Computationally, this is straightforward as we simply take the dot product of each point in the object with $\mathbf{d}$ and return the point that generates the largest dot product.

If an object contains numerous (or infinite) number of points, then the support function needs to be defined in a more appropriate manner. As an example, consider a sphere $\mathscr{S}$ with radius R and center at point C. The support function would then be

$$\text{support}(\mathscr{S},\mathbf{d}) = \mathbf{C} + R\frac{\mathbf{d}}{|\mathbf{d}|}. \tag{6}$$

In this case, the support function has a rather simple form. Other objects may have more complex support functions.

The support functions for Msum and Mdiff can become quite complex as they involve the sum and difference of arbitrary objects. Fortunately, there are simpler ways of computing the support functions. For Msum, we note that the support function always returns a point farthest in the given direction. Since Msum involves adding points from the 2 sets, we can obtain the farthest point in Msum by getting the point farthest along the direction in $\mathscr{A}$ and adding the point farthest along in $\mathscr{B}$.

In other words,

$$\text{support}(\text{Msum}(\mathscr{A},\mathscr{B}),\mathbf{d}) = \text{support}(\mathscr{A},\mathbf{d}) + \text{support}(\mathscr{B},\mathbf{d}) \tag{7}$$

where $\mathbf{d}$ is the direction.

Unfortunately, the support function for Mdiff is not as simple. We cannot go as far as possible in set $\mathscr{A}$ and subtract the farthest point in $\mathscr{B}$. The resulting point may not be the farthest point in $\text{Mdiff}(\mathscr{A},\mathscr{B})$. Instead, we use

$$\text{support}(\text{Mdiff}(\mathscr{A},\mathscr{B}),\mathbf{d}) = \text{support}(\mathscr{A},\mathbf{d}) - \text{support}(\mathscr{B},-\mathbf{d}) \tag{8}$$

That is, we go as far as we can along $\mathbf{d}$ in set $\mathscr{A}$ and subtract off the point in $\mathscr{B}$ that is farthest in the opposite direction, $-\mathbf{d}$.

As an example, we consider the sets shown in Fig. 3, and compute $\text{support}(\text{Mdiff}(\mathscr{C},\mathscr{D}),\hat{\mathbf{x}})$. $\text{support}(\mathscr{C},\hat{\mathbf{x}})$ returns a vector to 1 of the points on the $x=-1$ line. $\text{Support}(\mathscr{D},-\hat{\mathbf{x}})$ returns a vector to 1 of the points on the $x=1$ line. For both cases, multiple points could be selected, so the support functions for the sets are free to choose any of them. Regardless, subtracting the two vectors gives a vector to a point with $x=-2$, which, as shown in Fig. 3, lies farthest in the $\hat{\mathbf{x}}$ direction of $\text{Mdiff}(\mathscr{C},\mathscr{D})$.

GJK uses support functions to choose points in Mdiff without having to compute Mdiff itself. This allows GJK to be used for any convex object provided the object has a support method.

2.5 Simplices

A simplex is the convex generalization of a triangle to an arbitrary number of dimensions. A 0-simplex consists of a single point. A 1-simplex is the line segment between the 2 endpoints. A 2-simplex is a triangle among 3 points. A 3-simplex is a tetrahedron. In general, a d-simplex has d dimensions and consists of $d+1$ points. Importantly, a simplex is always the convex hull of the given number of vertices.

GJK uses simplices to help determine if the origin is located within Mdiff. GJK constructs a simplex by using the support function to select points along the convex hull of Mdiff. Since the resulting simplex has a simple form (a point, line segment, triangle, or tetrahedron) GJK can quickly determine if the origin is contained in the simplex or not. Since the theorem of Carathéodory guarantees that any point can be

contained within a convex hull of, at most, 4 points (for 3 dimensions), GJK only needs a 0-, 1-, 2-, or 3-simplex.

3. GJK Algorithm

GJK determines if 2 objects, $\mathscr{A}$ and $\mathscr{B}$, intersect (or collide). We first provide a quick overview of how GJK operates and then proceed to a more in depth discussion.

3.1 Basic Operation of GJK

3.1.1 Overview of GJK

If $\mathscr{A}$ and $\mathscr{B}$ intersect, then their Minkowski Difference, Mdiff$(\mathscr{A},\mathscr{B})$, will contain the origin. Thus, GJK searches Mdiff to determine if the origin is contained within it. However, Mdiff could be very complex making its direct computation intractable. Instead, GJK makes use of the theorem of Carathéodory, which says that if a point P lies within any convex hull, then P also lies within a reduced convex hull consisting of, at most, 4 points (in 3 dimensions) of the larger convex hull. Using the theorem, GJK focuses on constructing simplices from the points on the convex hull of Mdiff. The simplices have very simple forms, being either a point (0-simplex), a line segment (1-simplex), a triangle (2-simplex) or a tetrahedron (3-simplex), which allow GJK to quickly make determinations about whether the origin lies within the simplices.

The problem would be extremely simple if the proper simplex were known at the start. Since the proper simplex is not known, GJK must iteratively update the simplex to use additional points while disregarding points that are no longer useful. GJK determines what points to add and what points to remove based on the direction from the current simplex to the origin.

3.1.2 Two Examples of GJK Operating

As an example, we consider two Mdiff sets shown in Fig. 4. We first consider the set on the left, which was made by two intersecting objects, and thus includes the origin. GJK begins by selecting a point on the convex hull of the Mdiff by using the support function. That leaves a question as to what direction should be passed to the function. Since GJK is just beginning, any direction will be satisfactory, so we simply choose the x-direction. The support function returns point A, which GJK adds to the simplex. Since A is the only point in the simplex, GJK now has a 0-

simplex. To locate the next point, GJK chooses a direction that points from the 0-simplex toward the origin. The support function returns the point E, which is added to the simplex, creating a 1-simplex consisting of $\overline{AE}$, the line segment from A to E. Note that the point F was not returned by the support functions because E lies slightly farther.

Fig. 4 Two example Mdiff **sets demonstrating how GJK operates. The set on the left includes the origin while the set on the right does not (the origins are indicated by the small plus signs). The dashed lines indicate the segments added to the simplex by GJK.**

GJK next chooses a direction from the 1-simplex toward the origin. The support function chooses point G, which GJK adds to the simplex, forming a 2-simplex (a triangle). Note that point F was not selected. GJK chose a direction from the segment $\overline{AE}$, not from either of the end points. The direction from the segment toward the origin is not strictly downward. Instead, the direction includes a horizontal component, which causes the support function to choose G. Section 3.2.2 shows how to compute this direction.

GJK now has a 2-simplex consisting of the triangle formed by the points A, E, and G. Since the origin is not within the triangle, GJK has to select another point. In 2 dimensions, the 2-simplex is the largest simplex required, so GJK removes a point before adding a new point. Given the example, the origin lies on the other side of the segment $\overline{EG}$, so point A is no longer required. GJK removes A from the simplex, creating a 1-simplex consisting of the points E and G. GJK then chooses the next point by moving from the 1-simplex toward the origin and obtains point F from the support function. GJK adds F to the simplex, creating a new 2-simplex consisting of E, G, and F. The origin is contained within the simplex, causing GJK to stop and report an intersection.

Now consider the Mdiff set shown on the right-hand side of Fig. 4. This set does not contain the origin, indicating that the objects do not intersect. GJK proceeds in a similar fashion as above. The first point added is again A. Then GJK moves from A toward the origin and obtains point E. GJK now has a 1-simplex consisting of the ling segment from A to E. So far nothing is different than the earlier example. Now, however, GJK moves from the line segment $\overline{AE}$ toward the origin, which returns point F. GJK now has a 2-simplex with points A, E, and F. Since this 2-simplex does not contain the origin, GJK removes point A and tries again to find another point. No further points can be added since no points between the origin and the segment $\overline{EF}$ exist in Mdiff. GJK cannot find the origin within Mdiff and so reports no intersection.

These two examples demonstrate the operation of GJK as well as how GJK terminates. GJK operates by calling the support functions to select points on the convex hull of Mdiff$(\mathscr{A},\mathscr{B})$. GJK determines the direction to pass to the support function by considering the current simplex and the origin. GJK determines the feature (point, line segment, or triangle face) of the simplex that lies closest to the origin. Any point not involved in that feature is removed from the simplex. GJK then moves from the remaining points toward the origin to locate the next point to add.

3.1.3 Termination Conditions

GJK terminates either when the origin is located, or when GJK runs out of points to add to the simplex. The latter condition can be tricky to implement because the support functions are not required to return unique points. If the set contains numerous points all equally distant along some direction, the support function is free to return any of them. Thus, we may not be able to determine whether we have run out of points since the support function could continue to supply new and different points.

We require a better termination condition in the event that the origin is not contained within the simplex. We note that the support functions return a point farthest in the given direction. If we choose a direction that points toward the origin and move as far as we can in that direction, then the support function will return a point that will be on the other side of the origin if the origin is contained within Mdiff. That is, we always cross over the origin, if the origin lies within Mdiff. We see this in left side of Fig. 4, where we crossed the origin each time we added a new point. The origin was crossed when moving from A to E, from $\overline{AE}$ to G, and again when moving from

$\overline{EG}$ to F. By contrast, for the right side of Fig. 4, we did not cross the origin when moving from A to E since the origin was not within Mdiff.

To see how we can check for this condition, we examine moving from point A to point E for the examples in Fig. 4. A closeup of the situation is shown in Fig. 5. GJK moved from point A along direction **d**. The support function in both cases returned point E.

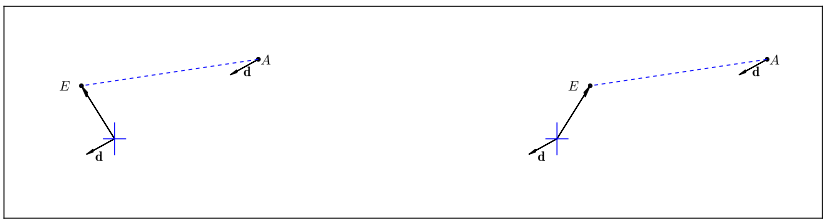

Fig. 5 Closeup view of moving from point A to point E for the examples from Fig. 4. For both cases, GJK computed the direction d to search for a new point, and, through the support function, found point E. On the left, the origin was crossed while moving from A to E, while on the right the origin was not crossed.

The left side of Fig. 5 shows the that the origin was crossed while moving from point A to point E, while the right side shows the case where the origin was not crossed. In both cases, we can draw the vector **E**, which points from the origin to point E. From the figure, we see that when the origin is crossed, **d** has a positive projection along **E**. On the other hand, if the origin is not crossed, then **d** has a negative projection along **E**.

This insight provides us with the termination condition. At each iteration, GJK computes a direction, **d**, to search for a new point to add to the simplex. **d** is passed to the support function, which returns a new point P. If

$$\mathbf{P} \cdot \mathbf{d} < 0 \tag{9}$$

then the origin was not crossed when moving to P and therefore, the origin is not located within Mdiff. GJK can report no intersection for this case.

Note that the condition given in Eq. 9 uses the less-than operator and not less-than-or-equal. It is possible that **d** is perpendicular to **OP**, in which case, $\mathbf{OP} \cdot \mathbf{d} = 0$. For this case, the origin could be located on the edge of Mdiff, inside of Mdiff, or on

the outside of Mdiff. We cannot say for certain which case applies, so GJK cannot terminate yet.

Finally, we note that this termination condition requires a simplex with at least 1 point in it already. In order for the concept of crossing the origin to make any sense, we must start from a point on the outside of Mdiff and move to another point in the direction of the origin. Thus, the simplex must have 2 points for the condition to be applied. Generally, this is only an issue at the very beginning of the algorithm when we have not added any points to the simplex yet.

3.1.4 GJK Algorithm

With the above discussions in mind, we now present algorithm 1, the basic GJK algorithm. The algorithm begins by setting the search direction to be the x direction and creating an empty simplex. The algorithm next enters main loop. On each iteration, GJK uses the search direction to find a new point on the convex hull of Mdiff, and adds that point to the simplex. If the origin is not crossed by moving to this new point, then GJK reports no intersection. Otherwise, the simplex is processed to determine if the origin is contained inside. If so, then GJK reports that the 2 objects intersect. If not, then GJK gets a new simplex and search direction to use in the next iteration. We discuss how to process the simplex in Section 3.2.

The GJK algorithm terminates when either of the terminating conditions discussed in Section 3.1.3 are met. The algorithm also terminates after 20 iterations have been performed, which we added to ensure that the algorithm would always terminate. The value of 20 is somewhat arbitrary. In our experience, GJK generally finds an answer within a handful of iterations and rarely needs more than 10. We also found that if GJK got to 20 iterations, there was usually a problem or bug in the code that needed to be solved. For now, we leave the iteration cap in the algorithm to ensure the algorithm always terminates.

The top level algorithm makes use of a few additional methods: add(), num_points(), process(), contains_origin(), and get_next(). We do not provide any pseudo-code for these methods as they involve book keeping details that detract from the overall algorithm. Nonetheless, their operations should be clear. add() simply stores the point in whatever data structure the simplex uses. num_points() returns the number of points in the simplex. process() calls the appropriate process methods given below. contains_origin() returns true if the simplex contains the origin. get_next()

```
def collision(𝒜, ℬ):
    niters = 0
    done = False
    res = False
    dir = [1, 0, 0]
    simp = Simplex()
    while not done and niters < 20:
        niters += 1
        P = 𝒜.support(dir) − ℬ.support(−dir)
        simp.add(P)
        if simp.num_points() > 1 and OP · dir < 0:
            res = False
            done = True
        simp.process()
        if simp.contains_origin():
            res = True
            done = True
        simp, dir = simp.get_next()
    return res
```

Algorithm 1: Basic GJK algorithm

returns the simplex and direction to use in the next iteration of GJK.

3.2 Simplex Processing

At each iteration of the GJK algorithm, GJK adds a point to the current simplex and then processes the simplex. As we saw in the examples in Section 3.1.2, processing the simplex involves determining if the origin is inside the simplex. If so, then there is no further action required. Otherwise, GJK examines the simplex to remove any unnecessary points and then computes a new direction to search for the next point to add.

We determine if the origin is inside the simplex and which points are unnecessary through the same general process. For each simplex, we examine each geometrical feature (i.e., vertex points, line segments, and triangle faces) one by one. Some features can immediately be eliminated from consideration simply due to the way the simplex was constructed. For each remaining feature, $\mathscr{F}$, we compute a direction, $\mathbf{d}$, that points from $\mathscr{F}$ and points away from the interior of the simplex. Usually $\mathbf{d}$ is the vector normal to $\mathscr{F}$, but not always. We next compute a vector, $\mathbf{FO}$, that points from $\mathscr{F}$ to the origin. If $\mathbf{d}$ has a positive projection along $\mathbf{FO}$, i.e., $\mathbf{FO} \cdot \mathbf{d} > 0$, then

we conclude that the origin lies outside the simplex and that $\mathscr{F}$ is the feature that we need. Any point in the simplex that is not part of $\mathscr{F}$ is no longer needed and can be removed the simplex. Thus, the simplex for the next iteration simply contains $\mathscr{F}$. The search direction for the next iteration is **d**.

If no such feature can be found, then the origin lies within the simplex, or possibly on one of the features themselves. Regardless, we conclude that the origin is contained with the simplex, which means that we have discovered that the 2 objects intersect.

Simplex processing can be performed by using dot and cross products appropriately applied to each feature, which makes the process is more geometrical in nature. By contrast, the original GJK paper[1] determined which feature was closest to the origin through a very algebraic procedure. The geometrical approach is generally simpler to implement and more intuitive than the algebraic approach. However, we must be extremely careful in how the direction **d** is computed from each feature $\mathscr{F}$. If **d** is computed incorrectly, then GJK may not be able to determine if the origin is located within a simplex. A number of the bugs we discovered during the development process were due to slight mistakes in computing the directions.

We also remark that the search direction computed for the next iteration of GJK is always from a feature and not from a point. (The only exception being the second iteration of GJK because the first iteration creates a 0-simplex.) This can have some surprising results, as we saw in Section 3.1.2 when examining the left side of Fig. 4. After we added point E to the simplex, we next added point G and not F, despite the fact that F would appear to be closer to the origin and E. The reason for this is that the direction computed was from the line segment feature $\overline{AE}$ and not the point E itself. We discuss this more when we handle the 1-simplex below.

When discussing each simplex type, we label the points in the simplex as A, B, C, and D. We adopt the convention that A is always the point that was most recently added to the simplex. This convention is critical, as it allows us to skip certain checks that we would otherwise need to perform. The ordering of the other points is less critical, though we generally make B the second most recently added point, C the third most recently added point, and D the first point added. As we discuss below, there are 2 instances where this ordering is modified: in the 2-simplex processing and in the 3-simplex. Regardless, A is always the most recently added point.

Finally, for each simplex type, we present pseudo-code for processing that simplex. The pseudo-code uses the notation given in Section 2, e.g., **AB** refers to the vector from point A to point B, $\mathbf{AB} \times \mathbf{AC}$ is the vector cross product, and so on. The syntax of the pseudo-code is an approximate object-oriented syntax where "self" refers to the instance variable (C++ programmers would use "this"). We present the pseudo-code for processing each simplex as methods of the simplex class.

3.2.1 Processing a 0-simplex

A 0-simplex consists of a single point, A, that lies on the convex hull of Mdiff. Since this vertex point is the only feature in the simplex, there are no further checks to be performed. The simplex for the next iteration contains this point. The only direction we can move is back from A toward the origin. Since **A** is the vector from the origin to A, $-\mathbf{A}$ must be the direction we need to move to come back to the origin.

A very rare case, although one that may still happen, is that point A may actually be the origin. In order for the origin to lie on the exterior of Mdiff, the 2 objects must intersect at exactly 1 point. Whether this contact constitutes an intersection depends on the application. Some applications regard such contact as a collision while others do not. We regard such contact as a collision, and, thus, need to check for the case where A is the origin. For applications that do not consider this case to be an intersection, the algorithm for processing the 0-simplex would need to be adjusted, as would some portions of the steps for processing the other simplices. With this in mind, we present algorithm 2 to handle the processing of a 0-simplex.

```
def process_0simplex(self):
    if A is the origin:
        mark self as containing the origin
    else:
        self.next_direction = −A
        self.next_simplex = self
```

Algorithm 2: Processing a 0-simplex

3.2.2 Processing a 1-simplex

A 1-simplex consists of the line segment between the 2 points, A, and B, where A was the most recent point added. There are 3 features, the point A, the point B, and the line segment $\overline{AB}$. These features divide the space around the simplex into 4 different regions, as shown in Fig. 6. Region R_1 lies beyond B. Region R_2 lies

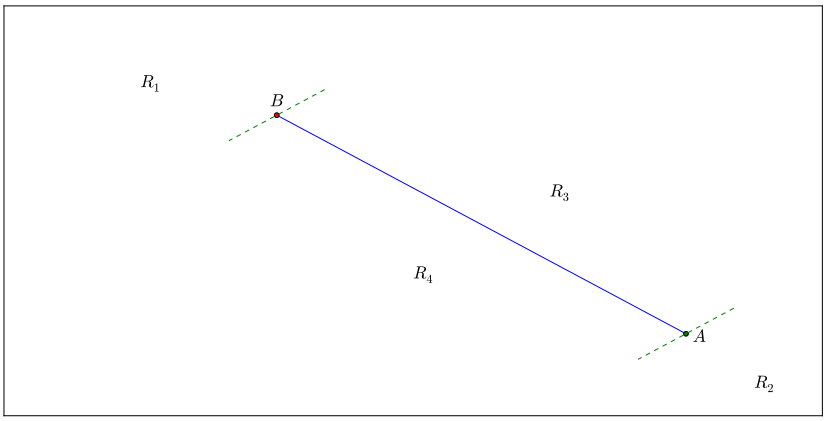

Fig. 6 Sketch of a 1-simplex, which is a line segment. The origin could be in any of the 4 indicated regions.

beyond A. Regions R_3 and R_4 lie on either side of the line segment. The origin could lie in any of these regions or on the line segment itself.

We need to check each of the regions for the origin. We can immediately eliminate R_1 and R_2 from consideration. At the beginning of this iteration, GJK had a 0-simplex containing the point B. GJK moved away from B toward the origin and found the point A. Thus, the origin cannot be in region R_1 since GJK moved away from that region already. The origin also cannot be in R_2. If the origin was in R_2, the GJK would have terminated since going from B to A did not cross over the origin.

That means the origin is located in either R_3 or R_4. We handle both simultaneously. $\mathbf{AB}$ is the vector from A to B while $\mathbf{AO}$ is the vector from A to the origin. Those 2 vectors define a plane whose normal is given by $\mathbf{AB} \times \mathbf{AO}$. If we cross the normal into $\mathbf{AB}$ again, we obtain a vector that points from the line segment toward the origin. Thus, the direction we want is given by $\mathbf{AB} \times \mathbf{AO} \times \mathbf{AB}$. Note that it does not matter if the origin lies in R_3 or R_4 since both cases result in the same expression for the direction.

Here we could ask why choose $\mathbf{AB} \times \mathbf{AO} \times \mathbf{AB}$ as the new direction to check, as opposed to simply choosing $\mathbf{AO}$ since $\mathbf{AO}$ points directly to the origin. The reason is that $\mathbf{AB} \times \mathbf{AO} \times \mathbf{AB}$ is perpendicular to the line segment, but $\mathbf{AO}$ may not be. In fact, $\mathbf{AO}$ could be located very close to $\mathbf{AB}$. If we move in that direction, we could

obtain B as our point to use for the next iteration. This would cause a ping-pong like effect where GJK continually bounces between selecting B, then A, then B, then A, and so on. Since we determined that the origin lies closest to the line segment and not either of the vertex points, we require a direction from the line segment towards the origin. The only useful direction would be $\mathbf{AB} \times \mathbf{AO} \times \mathbf{AB}$.

We also need to consider 2 boundary cases: A could be the origin itself or the origin could lie on the line segment $\overline{AB}$.

- If A is the origin, then $\mathbf{AO}$ is zero, meaning that $\mathbf{AB} \times \mathbf{AO} \times \mathbf{AB} = 0$.
- If the origin lies on the segment $\overline{AB}$, then $\mathbf{AB} \times \mathbf{AO} = 0$ since the 2 vectors would be parallel.

Thus, both of these cases require checking if $\mathbf{AB} \times \mathbf{AO} \times \mathbf{AB}$ is zero. Some implementations do not check for these cases.[4,6–9] Not handling these cases forces GJK to use a direction of zero to search for the next point to add. This could cause problems when used in the support functions since support functions are to return a point maximally far in the given direction. The point maximally far in the zero direction is an undefined concept, so the support functions may act in unknown ways when given a zero vector. An example of this is the support function for a sphere given in Eq. 6. If given a zero direction, that support function would return the center of the sphere and not a point on the edge of the sphere. This would cause GJK to consider points that are not on the exterior of Mdiff, which violates our assumptions. We could fix this behavior by demanding the support functions check for a zero direction, but we feel it is better to simply handle the case in our implementation of GJK.

```
def process_1simplex(self):
    dir = AB × AO × AB
    if dir is zero:
        mark self as containing the origin
    else:
        self.next_direction = dir
        self.next_simplex = self
```

Algorithm 3: Processing a 1-simplex

Algorithm 3 shows our implementation. Since the line segment $\overline{AB}$ is the feature closest to the origin, we require both points in the new simplex.

3.2.3 Processing a 2-simplex

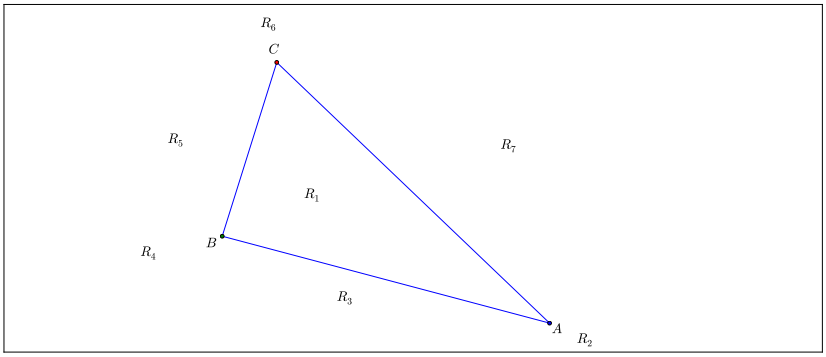

Fig. 7 Sketch of a 2-simplex and the regions created around it.

A 2-simplex is considerably more complex to handle than either the 1-simplex or 0-simplex. The 2-simplex forms a triangle in space, which contains 7 features: 3 vertex points, 3 line segments, and the triangle face itself. Each of these features corresponds to a particular region in space, as shown in Fig. 7. We label the vertices of the triangle, A, B, and C, again with the understanding that A was the last point added to the simplex. The 7 regions are R_1, which is the interior of the triangle; R_2, R_4 and R_6, which are the regions beyond the vertices; and R_3, R_5 and R_7, which lay beyond the line segments between each pair of vertices.

We now need to determine which region the origin lies in. Of the 7 possible regions, we immediately eliminate 3 from consideration. In the previous iteration, GJK had a 1-simplex consisting of the points B and C. GJK sought a new point by searching from $\overline{BC}$ toward the origin, resulting in point A. Thus, in searching for a point toward the origin, GJK moved away from R_4, R_5 and R_6, so the origin cannot be in those regions.

We can also eliminate region R_2 from consideration. If the origin were in R_2, then moving from $\overline{BC}$ to point A would not have crossed the origin, which means the top level GJK algorithm would have terminated already. The origin therefore must be located in regions R_3, R_7, or R_1.

Since R_3 and R_7 are regions that lie beyond line segments, we would expect their handling to be similar to that of a 1-simplex. Once again, we define a normal to the

line segment and check the region for the origin. However, there are important differences from the 1-simplex case since $\overline{AB}$ and $\overline{AC}$ come from a triangle. We cannot simply use the same procedure from 1-simplex case because we must differentiate between the cases of the origin lying in R_3, or R_7 or R_1.

Our task is to define a normal from $\overline{AB}$ that points into R_3 and a normal from $\overline{AC}$ that points into R_7, and then check each region. In 3 dimensions, there are an infinite number of vectors normal to either line segment. However, only the normals in the plane of the triangle are useful. Each segment has 2 such (unit) normals, pointing outward from, and inward to R_1. We determine these normals by first computing $\mathbf{ABC}_n$, which is the normal to the triangle,

$$\mathbf{ABC}_n = \mathbf{AB} \times \mathbf{AC} \tag{10}$$

The normals to each of the line segments are then defined as

$$\mathbf{AB}_n = \mathbf{AB} \times \mathbf{ABC}_n \qquad \mathbf{AC}_n = \mathbf{ABC}_n \times \mathbf{AC} \tag{11}$$

Carefully note the order in which the vector cross products are computed. We want the normals to the line segments to point away from the interior of the triangle. The normals defined in Eq. 11 obey this convention. If the order of the vectors are flipped in the cross products, then the resulting normals will point toward the interior of the triangle, which will cause GJK to compute incorrect results. Our experience is that computing incorrect normals is a common way of introducing bugs into the code, which can be very difficult to find.

With the normals correctly defined, we determine if the origin lies in R_3 by checking the condition

$$\mathbf{AB}_n \cdot \mathbf{AO} > 0. \tag{12}$$

If Eq. 12 is true, then the origin lies somewhere in R_3, so we no longer need point C in our simplex. We now need a direction to search for the next point. We compute that direction as

$$\mathbf{AB} \times \mathbf{AO} \times \mathbf{AB}, \tag{13}$$

which is analogous to the direction used for the 1-simplex case. Note that the direction we need to move is not given by $\mathbf{AB}_n$. In 2 dimensions, we could use $\mathbf{AB}_n$ as our new direction, but in 3 dimensions the origin may lie above (or below) the

plane of the triangle. The direction in Eq. 13 takes this into account, whereas $\mathbf{AB}_n$ does not. The check for R_7 is identical, except that C is used instead of B.

For completeness in handling R_3 and R_7, we note that we found a reference[3] that advocated checking $\mathbf{AB} \cdot \mathbf{AO} > 0$ and $\mathbf{BA} \cdot \mathbf{BO} > 0$ in addition to Eq. 12 for R_3 (and similar for R_7). In general, we do not feel these extra conditions are necessary as they are almost always satisfied. In order for the 2-simplex to be processed, we must have crossed the origin when moving to point A, as per the termination condition in GJK algorithm (algorithm 1). That guarantees that $\mathbf{AB} \cdot \mathbf{AO} > 0$ if A is the point exactly on the direction we moved from B. We note that the support functions do allow for some "wiggle" room in that they do not have to report a point exactly in the direction. However, the support functions still return a point approximately in the direction, which is usually enough to avoid these extra checks. Nonetheless, there may be some peculiar cases that require these extra checks, though we never encountered any in our experience with GJK. Other references[4–9] omit the extra checks entirely.

If the origin is not located within R_3 or R_7, then it must be located within R_1, which is the interior of the triangle. In 2 dimensions, this would be enough to report that the 2 objects intersect. However, in 3 dimensions, the origin may lie in, above, or below the plane of the triangle, each of which needs to be handled separately.

- If $\mathbf{ABC}_n \cdot \mathbf{AO} = 0$, then the origin is in the plane of the triangle and located in the interior of the triangle. This means we have found the origin within Mdiff and can report an intersection.
- If $\mathbf{ABC}_n \cdot \mathbf{AO} > 0$, then the origin lies above the plane of the triangle. In this case, the simplex stays the same and we search for the next point in the direction of $\mathbf{ABC}_n$.
- If $\mathbf{ABC}_n \cdot \mathbf{AO} < 0$, then the origin lies below the plane of the triangle. We thus need to move in the direction of $-\mathbf{ABC}_n$ to find our next point. Unlike the earlier case where the origin lay above the plane, we need to adjust the simplex slightly. As discussed in the next section, the processing of a 3-simplex requires that $\mathbf{AB} \times \mathbf{AC}$ point in the direction of the next point added. This is not true if we move below the plane of the triangle. To make it true, we interchange A and B in our simplex ordering.

We also need to consider the possibility that A might be the origin, or that the origin might lie on 1 of the line segments. As it turns out, none of these cases requires any special handling.

- If A is the origin, then $\mathbf{AO}$ will be zero, which means all the dot products involving $\mathbf{AO}$ will also be zero. By the above discussion, the origin would lie in R_1, and, since $\mathbf{ABC}_n \cdot \mathbf{AO} = 0$, the origin would lie within the plane of the triangle. We thus report an intersection.
- If the origin lies on any of the edges, then the dot product of $\mathbf{AO}$ with the normal to that edge will be zero. The dot product with the other edge normal will be negative, so the origin again lies in R_1. Since the origin is in the plane of the triangle, we report an intersection.

With these discussions in mind, the algorithm for handling the 2-simplex case is shown in algorithm 4.

3.2.4 Processing a 3-simplex

A 3-simplex is a tetrahedron consisting of 4 triangle faces. As such, we would expect that processing a 3-simplex is similar to processing a 2-simplex. There are still a few slight differences, however.

Figure 8 shows a 3-simplex, which is a useful diagram for the rest of this section. GJK started the previous iteration with a 2-simplex that consisted of the points B, C and D. During that iteration, point A was added to the simplex to create the 3-simplex that we now need to examine. Importantly, we make the assumption that the point A lies in the direction of $\mathbf{BC} \times \mathbf{BD}$. If this assumption is violated, then our processing of the 3-simplex will run into difficulties.

To ensure this assumption is met, we need to reexamine how the 3-simplex was created from a 2-simplex. During the processing of the 2-simplex, GJK chose the next search direction to be above or below the plane of the triangle to find point A. If the direction was above the plane, then the assumption of $\mathbf{BC} \times \mathbf{BD}$ pointing toward A is automatically satisfied. If the direction was below the plane of the triangle, then A points in the direction of $-\mathbf{BC} \times \mathbf{BD}$, which violates our assumption. However, if we exchange the order of any 2 of the points, say B and C, then the direction changes. This is the reason we changed the order of the points in the 2-simplex

```
def process_2simplex(self):
    ABC_n = AB × AC
    AC_n = ABC_n × AC
    AB_n = AB × ABC_n
    if AB_n · AO > 0:
        # Region 3
        self.next_direction = AB × AO × AB
        self.next_simplex = Simplex([B, A])
    elif AC_n · AO > 0:
        # Region 7
        self.next_direction = AC × AO × AC
        self.next_simplex = Simplex([C, A])
    else:
        # Region 1; Origin could be below, or
        # above, or in the plane of the triangle
        v = ABC_n · AO
        if v is zero:
            # In the plane of the triangle
            mark self as containing the origin
        elif v > 0:
            # Above the plane of the triangle
            self.next_direction = ABC_n
            self.next_simplex = self
        else:
            # Below the plane of the triangle
            # Note: we change the order of the simplex. See
              the
            # handling of a 3-simplex for why.
            self.next_direction = −ABC_n
            self.next_simplex = Simplex([C, A, B])
```

Algorithm 4: Processing a 2-simplex

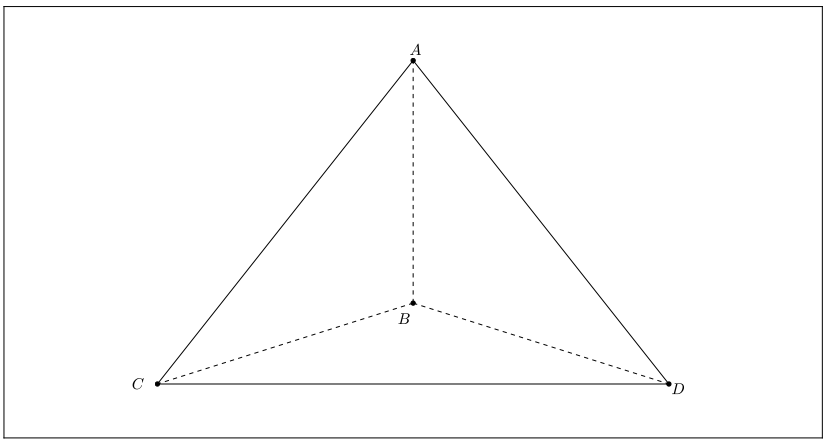

Fig. 8 Sketch showing a 3-simplex (a tetrahedron). The point A lies above the plane determined by the triangle $\triangle BCD$. The point B lies behind the $\triangle ACD$ triangle face.

when the search was chosen to point below the triangle. By switching 2 of the points, we inverted the direction of the normal and ensured this assumption is valid.

With our tetrahedron properly oriented, we now need to determine which region the origin could lie in. We have 15 features to examine: 4 vertices, 6 line segments, 4 triangle faces, and the tetrahedron itself. As we have done previously, we immediately eliminate some features from consideration. The previous iteration used a 2-simplex that had the points B, C, and D, and moved toward the origin to find the point A. Thus, we can eliminate the regions related to the points B, C, and D, the line segments $\overline{BC}$, $\overline{CD}$ and $\overline{DB}$, and the triangle $\triangle BCD$. Further, we can eliminate the region beyond A. If the origin was there, then GJK would have terminated already since moving from $\triangle BCD$ to A did not cross the origin. This leaves us with 7 features: the tetrahedron, 3 triangle faces, and 3 line segments.

We process those features by first determining if the origin lies above any of the triangle faces, $\triangle ABC$, $\triangle ACD$, and $\triangle ADB$. If so, then we conclude that the origin is outside the tetrahedron, but may possibly still lie within Mdiff. Thus, a new simplex must be constructed and a new search direction chosen. The new simplex is either the triangle face itself, or an edge, whichever feature is closer to the origin. If the origin does not lie outside of any of the faces, then the origin is contained within the tetrahedron and we can report that the objects intersect.

We determine if the origin lies above a triangle face by computing the dot product of **AO** with the normal to the triangle face. If the dot product is positive, then the origin lies closest to that face. For this procedure to work, we need to define the normals to each face so that the normals point outward from the tetrahedron. We define the normals as

$$\mathbf{ABC}_n = \mathbf{AB} \times \mathbf{AC} \qquad \mathbf{ACD}_n = \mathbf{AC} \times \mathbf{AD} \qquad \mathbf{ADB}_n = \mathbf{AD} \times \mathbf{AB}. \quad (14)$$

Note carefully how the normals are defined in Eq. 14. The order of the vectors in the cross products were chosen to ensure the normals pointed outward. Choosing any other order causes at least 1 of the normals to point inward. Further, the definitions of the normals are consistent with the tetrahedron laid out in Fig. 8. This is why we made the assumption of A lying in the direction of $\mathbf{BC} \times \mathbf{BD}$. If A does not lie in that direction, then normals would have to be defined differently.

We now have to determine how to handle each triangle face. The procedure is identical for each face, so we only need to discuss the process for a single face. Suppose that $\mathbf{AO} \cdot \mathbf{ABC}_n > 0$, then the origin lies closest to the $\triangle ABC$ triangle. Right away, we know that point D is no longer needed and can be removed from the simplex. We now need to determine if the origin lies closer to an edge or to the face itself. This is almost identical to the 2-simplex handling discussed earlier. The only difference is that we do not need to check if the point lies below the triangle since we already know that it lies above it. That leaves us with 3 possibilities: the origin lies in the direction from the $\overline{AC}$ edge or the $\overline{AB}$ edge, or above the $\triangle ABC$ face itself.

As with the 2-simplex case, we distinguish between these cases by computing the relevant normals. For the edges, we define the normals as $\mathbf{AC}_n = \mathbf{ABC}_n \times \mathbf{AC}$ and $\mathbf{AB}_n = \mathbf{AB} \times \mathbf{ABC}_n$. The triangle face has normal $\mathbf{ABC}_n$, which we already computed. We check for, and handle, each of the 3 cases by checking the dot product of the normal with **AO**.

- If $\mathbf{AO} \cdot \mathbf{AC}_n > 0$, then the origin lies closer to the $\overline{AC}$ edge. We no longer need points B and D, so we create a 1-simplex consisting of A and C. The new direction to move is $\mathbf{AC} \times \mathbf{AO} \times \mathbf{AC}$.
- If $\mathbf{AO} \cdot \mathbf{AB}_n > 0$, then the origin lies closer to the $\overline{AB}$ edge. This is identical to the first case, except that B and C interchange.

- If neither of those cases is true, then the origin lies directly above the triangle face. We could check $\mathbf{AO} \cdot \mathbf{ABC}_n$ to confirm this, but we already computed this product. In this case, we create a new 2-simplex consisting of A, B, and C. The new direction to search is given by $\mathbf{ABC}_n$.

The processing of the triangles $\triangle ACD$ and $\triangle ADB$ is almost identical to the $\triangle ABC$ case except for changing which points are involved. There are, however, 2 caveats that need careful attention. The first caveat is that the edge normals are computed differently when using different triangle normals. Thus, $\mathbf{AC}_n$ is different when examining triangle $\triangle ABC$ than it is when examining triangle $\triangle ACD$.

The second caveat is that when creating a 2-simplex, we need to ensure that the points are ordered correctly in the simplex. If we decide to create a 2-simplex, then, in the next iteration, we add another point creating a 3-simplex. Consistent with our assumption above, we require that new point to lie in the direction of the cross products of the edges. Thus, we need to order the points to ensure that this assumption is upheld. This only affects the case for the $\triangle ADB$ triangle.

Finally, the origin could either be point A itself, or the origin could lie in 1 of the triangle faces, or the origin could lie in the plane of 1 of the triangles, but not in the face itself. These boundary cases do not require any special processing.

- If A is the origin, then **AO** will be zero and all the dot products we check will be zero. Thus, we consider that the origin lies inside the tetrahedron and report an intersection.
- If the origin lies within a triangle face, then the dot product of **AO** with that face's normal will be zero and the dot product with the other normals will be negative. Again, this causes us to consider the origin as being inside the tetrahedron.
- Lastly, if the origin lies in the plane of a triangle, but not in the triangle face itself, then the dot product of **AO** with the plane normal will still be zero. In this case, the origin will lies above 1 of the other triangle faces causing the dot product of **AO** with that normal to be positive, and so this case is handled by creating a new 2-simplex and moving on.

With that in mind, the algorithm for handling the 3-simplex is shown in algorithm 5.

```
def process_3simplex(self):
    # The 3-simplex consists of the points A, B, C, D.
    # We assume that A lies in the direction of BC × BD
    ABC_n = AB × AC
    ACD_n = AC × AD
    ADB_n = AD × AB
    if ABC_n · AO > 0:
        AC_n = ABC_n × AC
        AB_n = AB × ABC_n
        if AC_n · AO > 0:
            self.next_simplex = Simplex([C, A])
            self.next_direction = AC × AO × AC
        elif AB_n · AO > 0:
            self.next_simplex = Simplex([B, A])
            self.next_direction = AB × AO × AB
        else:
            self.next_simplex = Simplex([C, B, A])
            self.next_direction = ABC_n
    elif ACD_n · AO > 0:
        AD_n = ACD_n × AD
        AC_n = AC × ACD_n
        if AD_n · AO > 0:
            self.next_simplex = Simplex([D, A])
            self.next_direction = AD × AO × AD
        elif AC_n · AO > 0:
            self.next_simplex = Simplex([C, A])
            self.next_direction = AC × AO × AC
        else:
            self.next_simplex = Simplex([D, C, A])
            self.next_direction = ACD_n
    elif ADB_n · AO > 0:
        AB_n = ADB_n × AB
        AD_n = AD × ADB_n
        if AB_n · AO > 0:
            self.next_simplex = Simplex([B, A])
            self.next_direction = AB × AO × AB
        elif AD_n · AO > 0:
            self.next_simplex = Simplex([D, A])
            self.next_direction = AD × AO × AD
        else:
            self.next_simplex = Simplex([B, D, A])
            self.next_direction = ADB_n
    else:
        mark self as containing the origin
```

Algorithm 5: Algorithm for processing a 3-simplex

3.3 GJK Efficiency: Qualitative

GJK tends to be extremely efficient, both spatially and temporarily, when computing answers. There are no large data structures to process, nor any particularly computationally intensive steps in the algorithm. These factors make GJK quite efficient. Establishing absolute bounds on the spatial and temporal requirements is difficult because GJK relies upon external data structures (the objects being tested) and external support functions. These influences could cause GJK to vary wildly in how much space and time are required to test the intersection of 2 objects. Provided that these external influences are not too burdensome, GJK performs very efficiently.

Spatially, GJK only requires space for the simplex (ignoring the space for the objects). The simplex only needs, at most, four 3-D points; a new direction to search for points; a few state variables, such as a flag describing whether the simplex contains the origin; and space for a few temporary vectors used during the processing of each simplex type. None of these items are large, making the spatial requirements almost negligible.

Temporally, GJK requires a series of iterations to determine the result. The single most expensive operation in any of the iterations is computing vector cross products, which requires 6 multiplications and 3 subtractions. Processing a 3-simplex requires, at most, 7 cross products to be computed, which does not require much time on modern processors.

The number of iterations GJK requires is difficult to predict, as it depends on the objects involved and the initial direction chosen. GJK can usually figure out an answer within a handful of iterations, usually around 5–10. If the objects intersect, GJK generally requires at least 4 iterations in order to create the 3-simplex and determine if the origin is contained within. GJK may require a few more iterations to construct another 3-simplex, but usually that is the most required. If the objects do not intersect, then GJK is usually able to determine that within 2–3 iterations since GJK quickly discovers that the origin was not crossed when selecting a new point. Thus, we generally expect no more than about 10 iterations to determine an answer.

Algorithm 1 limited the number of iterations to 20 to ensure that the algorithm always terminates. Twenty was arbitrarily chosen based on simply doubling the

number of expected iterations. In our experience, we found that the only time GJK ever needed 20 iterations was when a bug was present in the code. After correcting any bugs that occurred, we never found GJK to require 20 iterations. We left the iteration bound in the algorithm just in case an unknown bug was encountered, although it may be possible for some exotic objects to require that many iterations to determine collisions.

Taken all together, we find that GJK is extremely efficient in time and space requirements. This makes GJK applicable in a wide variety of applications, especially the system effectiveness modeling that spurred our initial interest in GJK.

3.4 GJK Summary

GJK is an efficient algorithm for detecting the collision between 2 objects, $\mathscr{A}$ and $\mathscr{B}$. GJK reduces the intersection problem to the problem of determining if the origin lies within Mdiff$(\mathscr{A},\mathscr{B})$. GJK constructs a simplex consisting of points chosen from the convex hull of Mdiff. Each simplex divides the space bounded by Mdiff into several regions based on the geometrical features (vertices, line segments, triangle faces) of the simplex. Each region is checked to see if the origin may lie in that region. If so, then GJK creates a new simplex consisting of the points involved in the given feature, and then chooses a new direction to search for a new point from the convex hull of Mdiff. The direction chosen is usually perpendicular to feature itself. If no region contains the origin, then the origin lies within the simplex itself and GJK reports an intersection since Mdiff would therefore contain the origin. Of course, the origin may lie on a feature itself, which we treat as special cases. If GJK selects a new point, but does not cross over the origin to reach that point, then GJK stops since the origin is clearly not contained within Mdiff.

Algorithm 1 shows the basic GJK algorithm, with algorithms 2 – 5 showing how to process each type of simplex. The algorithms show that GJK is fairly simple and straightforward to implement. However, as the discussion about those algorithms indicates, numerous assumptions and conditions go into the algorithms that need careful attention. These assumptions and conditions are not often stated, which can result in hard to find bugs when implementing the algorithms. Those assumptions and conditions are summarized below:

- The support function always returns a point farthest in the given direction. This is undefined when a zero direction is used, so we need to ensure that the

support functions are never given a zero direction.

- The points added to the simplex always come from the convex hull of Mdiff. This condition is satisfied provided the support function operates correctly.
- In the algorithms, the point A is always the last point added to the simplex. The ordering of the other points in the simplex is less critical, but we generally maintain the order in which they were added.
- When we process the 2-simplex and 3-simplex, we need to compute normals that always point away from the simplex. The normals should never point toward the interior of the simplex.
- For a 3-simplex, the point A always lies in the direction of $\mathbf{BC} \times \mathbf{BD}$. Obeying this assumption may require slightly reordering the points in the simplex.

4. Verification of Implementation

After implementing the GJK algorithm, we sought to verify our implementation by comparing results against an algorithm that computes intersections among triangles,[10] which we label TriTri. TriTri shares nothing in common with GJK. We did not create our own implementation TriTri, choosing instead to use the implementation created by TriTri's authors.[10]

To make the comparison, we randomly generated 1 million pairs of triangles in 3 dimensions. For each pair, we used both our GJK implementation and TriTri to determine if the 2 triangles intersected. We counted the number of times our implementation and TriTri agreed, either by saying both triangles intersected or both did not, and counted the number of times the 2 disagreed.

Out of the 1 million pairs of triangles, our GJK implementation agreed with TriTri for all but 2 pairs of triangles. In both cases, our GJK implementation reported that the triangles intersected whereas TriTri reported they did not. Upon further investigating, we found that in both cases the triangles intersected at exactly 1 point along an edge. The intersection point was not any of the vertices, nor were the triangles coplanar. We believe that this might actually be a previously unknown bug in TriTri since TriTri reports that a pair of triangles intersect if they share a common vertex. Thus, TriTri clearly considers 2 triangles to be intersecting if they share exactly 1 point. Yet, for these 2 cases, TriTri could not detect the intersection.

This provides us with confidence that our implementation reports correct results. However, we cannot fully declare that our implementation to be correct. There could be an unknown bug lurking in either the algorithm or the implementation. As mentioned in the introduction, there are numerous difficulties in implementing GJK and implementations often disagree about small details.

5. Conclusion

We have presented a geometrical approach to the GJK[1] algorithm, which may be used to detect if 2 arbitrary convex objects intersect. The geometrical approach is generally simpler to implement and may be more intuitive for some readers. However, implementing the geometrical approach requires one to be extremely careful to make sure assumptions are not violated. When we first attempted to implement GJK, we experienced several difficulties because certain assumptions were never explicitly declared. Part of our purpose in writing this report was to lay out the various assumptions, why they are needed, and how to avoid common problems.

We compared our implementation of GJK against a program that computes the intersection of triangles and found good agreement, with disagreement in only 2 out of 1 million cases. In both cases, our implementation was determined to operate correctly.

We are currently using our implementation of GJK in our system effectiveness models. Within these models, we need to determine if 2 objects collide, where the objects could represent a vehicle, a threat against the vehicle, or a counter-munition designed to mitigate the threat, etc. As such, these objects have a wide variety of shapes and orientations. We have found GJK to be extremely fast and accurate in determining collisions among these objects.

6. References and Notes

1. Gilbert E, Johnson D, Keerthi S. A fast procedure for computing the distance between complex objects in three-dimensional space. IEEE Journal of Robotics and Automation 1988;4.

2. Breech B. A. Modeling system effectiveness of sensors and armors with mixing and matching components. Aberdeen Proving Ground (MD): Army Research Laboratory (US); 2015. Report No.: ARL-TR-7195.

3. Ericson C. The Gilbert-Johnson-Keerthi Algorithm. Notes published as part of a SIGGRAPH presentation. Available at `http://realtimecollisiondetection.net/pubs/SIGGRAPH04_Ericson_GJK_notes.pdf`, 2004.

4. `http://entropyinteractive.com/2011/04/gjk-algorithm/`, accessed 2014.

5. `http://www.codezealot.org/archives/88`, accessed 2014.

6. `http://physics2d.com/content/gjk-algorithm`, accessed 2014.

7. `http://programyourfaceoff.blogspot.com/2012/01/gjk-algorithm.html`, accessed 2014.

8. `https://mollyrocket.com/849`, accessed 2014.

9. `http://www.cs.sjsu.edu/faculty/pollett/masters/Semesters/Spring12/josh/GJK.html`, accessed 2014.

10. Möller T. A fast triangle-triangle intersection test. Journal of Graphics Tools 1997.

1 (PDF)	DEFENSE TECHNICAL INFORMATION CTR DTIC OCA
2 (PDF)	DIRECTOR US ARMY RESEARCH LAB RDRL CIO LL IMAL HRA MAIL & RECORDS MGMT
1 (PDF)	GOVT PRINTG OFC A MALHOTRA
4 (PDF)	RDRL WML A B BREECH W OBERLE L STROHM R YAGER

INTENTIONALLY LEFT BLANK.